Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web

Venkata Duvvuri, **Prashant Shenoy** and **Renu Tewari**†

Department of Computer Science, University of Massachusetts, Amherst, MA 01003 {dvshekar,shenoy}@cs.umass.edu

†IBM Research Division, T J. Watson Research Center, Hawthorne, NY 10532 tewarir@watson.ibm.com

Abstract— **In this paper, we argue that weak cache consistency mechanisms supported by existing web proxy caches must be augmented by strong consistency mechanisms to support the growing diversity in application requirements. Existing strong consistency mechanisms are not appealing for web environments due to their large state space or control message overhead. We focus on the lease approach that balances these tradeoffs and present analytical models and policies for determining the optimal lease duration. We present extensions to the http protocol to incorporate leases and then implement our techniques in the Squid proxy cache and the Apache web server. Our experimental evaluation of the leases approach shows that: (i) our techniques impose modest overheads even for long leases (a lease duration of 1 hour requires state to be maintained for 1030 leases and imposes an per-object overhead of a control message every 33 minutes); (ii) leases yields a 138-425% improvement over existing strong consistency mechanisms; and (iii) the implementation overhead of leases is comparable to existing weak consistency mechanisms.**

I. Introduction

A. Motivation

The growth of the Internet and the World Wide Web has enabled an increasing number of users to access vast amounts of information stored at geographically distributed sites. Due to the growing user population and the non-uniformity of information access, however, popular objects create server and network overload, and thereby significantly increase latency for information access [14]. Proxy caching is one popular approach to alleviate these drawbacks. In a proxy caching architecture, clients request objects from a proxy; the proxy services client requests using locally cached data or by fetching the requested object from the server. By caching frequently accessed objects, a proxy can reduce the load on network links and servers as well as client access latencies. A limitation, however, is that the proxy cache may store stale data.

To prevent stale information from being transmitted to clients, a proxy must ensure that locally cached data is consistent with that stored on servers. The exact cache consistency mechanism employed by a proxy depends on the nature of the cached data; not all types of data need the same level of consistency guarantees. Consider the following example.

Example 1: Online auctions. Consider a web server that offers online auctions over the Internet. For each item being sold, the server maintains information such as its latest bid price (which changes every few minutes) as well as other information such as photographs and reviews for the item (all of which change less frequently). Consider a proxy that caches this information. Clearly, the bid price returned by the proxy cache should always be consistent with that at the server. In contrast, reviews of items need not always be consistent, since a user may be willing to receive slightly stale information.

The above example shows that a proxy cache will need to provide different consistency guarantees for different types of data.

Most proxies deployed in the Internet today provide only weak consistency guarantees [9], [13]. Until recently, most objects stored on web servers were relatively static and changed infrequently. Moreover, this data was accessed primarily by humans using browsers. Since humans can tolerate receiving stale data (and manually correct it using browser reloads), weak cache consistency mechanisms were adequate for this purpose. In contrast, many objects stored on web servers today change frequently and some objects (such as newspapers and stock quotes) are updated every few minutes [1]. Moreover, the web is rapidly evolving from a predominantly read-only information system to a system where collaborative applications and program-driven agents frequently read as well as write data. Such applications are less tolerant of stale data than humans accessing information using browsers. These trends argue for augmenting the weak consistency mechanisms employed by today's proxies with those that provide strong consistency guarantees in order to make caching more effective.[1] In the absence of such strong consistency guarantees, servers resort to marking data as uncacheable, and thereby reduce the effectiveness of proxy caching. The design of efficient strong cache consistency mechanisms that can coexist with existing weak consistency mechanisms is the subject matter of this paper.

[1] Replacing existing weak consistency mechanisms altogether with strong consistency ones may impose more overheads than are necessary on applications that don't need strong guarantees. An incremental approach that augments existing mechanisms with those for strong consistency seems more practical and efficient.

B. Existing Cache Consistency Mechanisms: Benefits and Limitations

A cache consistency mechanism that always returns the results of the latest write at the server is said to be *strongly consistent*. Due to the unbounded message delays in the Internet, no cache consistency mechanism can be strongly consistent in this idealized sense. Hence, we relax our definition to the following: a mechanism that returns data that is never outdated by more than t time units with the version on the server is said to be strongly consistent, where t is the server to proxy delay at that instant and $0 < t \leq \infty$. Mechanisms that do not satisfy this property (i.e., can return stale data) are said to be *weakly consistent*.

Most existing proxies provide only weak consistency by (i) employing a server specified lifetime of an object (referred to as the *time-to-live (TTL)* value), or (ii) periodically *polling* the server to verify that the cached data is not stale [4], [9], [13]. In either case, modifications to the object before its TTL expires or between two successive polls causes the proxy to return stale data.

Strong consistency can be enforced either by *server-driven* mechanisms or *client-driven* mechanisms [16]. The former approach, referred to as *server-based invalidation*, requires the server to notify proxies when the data changes. This approach is optimal in the number of control messages exchanged between the server and the proxy (since messages are sent only when an object is modified). However, it requires the server to maintain per-object state consisting of a list of all proxies that cache the object; the amount of state maintained can be significant especially at popular web servers. Moreover, when a proxy is unreachable due to network failures, the server must either block on a write request until a timeout occurs, or risk violating consistency guarantees.

The client-driven approach, also referred to as *client polling*, requires that proxies poll the server on *every read* to determine if the data has changed [16]. Frequent polling imposes a large message overhead and also increases the response time (since the proxy must await the result of its poll before responding to a read request). The advantage, though, is that it does not require any state to be maintained at the server, nor does the server ever need to block on a write request (since the onus of maintaining consistency is on the proxy).

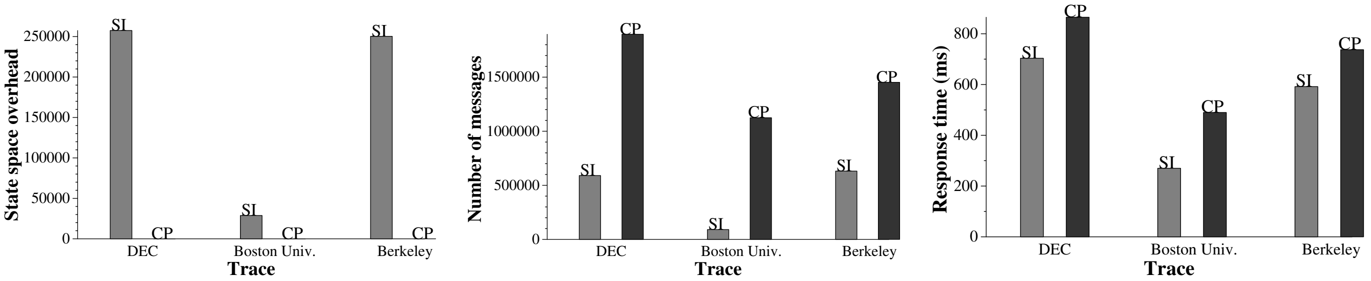
Server-based invalidation and client polling form two ends of a spectrum. Whereas the former is optimal in the number of control messages exchanged but may require a significant amount of state to be maintained, the latter is stateless but can impose a large control message overhead. Figure 1 quantitatively compare these two approaches with respect to (i) the server overhead, (ii) the network overhead, and (iii) the client response time. Due to their large overheads, neither approach is appealing for web environments. A strong consistency mechanism suitable for the web must not only reduce client response time, but also balance both network and server overheads.

One approach that provides strong consistency, while providing a smooth tradeoff between the state space overhead and the number of control messages exchanged, is *leases* [8]. In this approach, the server grants a lease to each request from a proxy. The lease duration denotes the interval of time during which the server agrees to notify the proxy if the object is modified. After the expiration of the lease, the proxy must send a message requesting renewal of the lease. The duration of the lease determines the server and network overhead. Smaller the lease duration, smaller is the server state space overhead, but larger is the number of control (lease renewal) messages exchanged and vice versa. In fact, an infinite lease duration reduces the approach to server-based invalidation, whereas a zero lease duration reduces it to client-polling. Thus, the leases approach spans the entire spectrum between the two extremes of server-based invalidation and client-polling.

The concept of a lease was first proposed in the context of cache consistency in distributed file systems [8]. Recently some research groups have begun investigating the use of leases for maintaining consistency in web proxy caches. The use of leases for web proxy caches was first alluded to in [3] and was subsequently investigated in detail in [16]. The latter effort has focused on the design of *volume leases*—leases granted to a collection of objects—so as to reduce (i) the lease renewal overhead and (ii) the blocking overhead at the server due to unreachable proxies. Other efforts have focused on extending leases to hierarchical proxy cache architectures [15], [17]. Thus, most of the research on leases to date has focused on the mechanisms for efficiently granting and renewing leases. The problem of determining the optimal lease duration so as to balance the tradeoff between the state space overhead and the control message overhead has not received much attention and is the focus of this paper.

C. Research Contributions

In this paper, we argue that weak consistency mechanisms supported by today's proxies must be augmented by strong consistency mechanisms to meet the growing diversity in application and user requirements. Since existing strong consistency mechanisms either impose a large state space overhead or a large control message overhead, we focus on the leases approach that balances these tradeoffs. We observe that the lease duration is the critical parameter that determines the efficiency of the leases algorithm and propose a number of techniques for determining the lease duration. First, we present analytical models that use constraints on the state space overhead and the control message overhead to compute an appropriate lease duration. Since these models are suitable only for scenarios where the load does not fluctuate rapidly, we then present a number of policies that enable a server to react to the fast time-scale variation in load. These policies require the lease duration to be computed afresh on each request, thereby enabling the server to immediately react to load fluctuations. Both techniques enable us to adapt the lease duration to the observed load,



(a) State Space overhead (b) Control Messages (c) Response time

Fig. 1. Efficacy of server-based invalidation and client polling for three different trace workloads (DEC, Berkeley, Boston University). The figure shows that server-based invalidation has the largest state space overhead; client polling has the highest control message overhead; server-based invalidation has the smallest response time.

albeit at different time scales; hence we collectively refer to our techniques as *adaptive leases*.

We have implemented the leases algorithm in the Apache web server and the Squid proxy cache. We present extensions made to the http/1.1 protocol to incorporate leases and then describe the details of our prototype implementation. Our implementation allows the proxy and the server to continue using existing weak consistency mechanisms and use the strong consistency provided by leases only when necessary.

Finally, we experimentally demonstrate the efficacy of the leases algorithm using trace-driven simulations and the prototype implementation. Our results show that (i) the dynamic lease computation policies allow a server to optimize either the state space overhead or the control message overhead depending on which factor is the bottleneck, and (ii) the state space and control message overhead imposed is modest even for relatively long leases (e.g., a lease duration of 1 hour imposes a state space overhead of 1030 leases and a per-object control message overhead of 0.0005 msg/sec—a 425% and 138% improvement over server-invalidation and client polling, respectively). Results from our prototype implementation shows that the overhead of computing and renewing leases is small (around 4ms, or 4.3% of the client response time) and is comparable to existing cache consistency mechanisms such as time-to-live values.

The rest of this paper is structured as follows. We present analytical models and adaptive policies for computing the lease duration in Sections III and IV, respectively. Section V discusses the details of our prototype implementation. Section VI presents our experimental results, and finally, Section VII presents some concluding remarks.

II. Adaptive Leases

Consider a web proxy that services user requests from its local cache, fetching requested objects from the server if necessary. Assume that the server and the proxy employ *leases* to provide strong consistency guarantees. Intuitively, a lease is a contract that gives its holder specific rights over property for a limited period of time [8]. In the context of web objects, a lease grants to its holder a guarantee that so long as the lease is valid the object will not be modified without prior notification. More formally, a lease for an object $\mathcal{O}$ consists of a pair (s, d) where s and d denote the start time and the duration of the lease, respectively, and the server agrees to notify the holder of all updates to the object within the interval $s \leq t < s + d$.

In such an environment, a proxy must hold a valid lease on a cached object before responding to a client read request.[2] The leases algorithm involves the following message exchange between the proxy and the server:

1. The first read for an object causes the proxy to send a lease grant request along with a GET request to the server; the server responds with the data along with a lease (or a lease denial).
2. Subsequent reads are served by the proxy from its local cache as long as the lease remains valid.
3. A read request after the expiration of a lease causes the proxy to send a lease renewal request along with an if-modified-since (IMS) request to the server; the server responds with a new lease (or a lease denial) along with either a not-modified message or the updated object.
4. Modifications to the object during the validity of its lease cause the server to send invalidation messages to all proxies holding a lease on the object; the server defers the update until it receives invalidate acknowledgements from all proxies or the lease duration expires.

Observe that, in addition to notifying the proxy of an update, the server can piggyback the update with the invalidation request [6]. Such piggybacking of modifications (also referred to as deltas [11]) can reduce the access latency for a subsequent read request without any significant increase in bandwidth requirements [10].

The crucial parameter that determines the efficiency of the lease algorithm is the lease duration d. By appropriately determining d, a server can balance the amount of state it needs to maintain and the number of control messages (lease renewal) exchanged. In what follows, we design techniques for determining the lease duration. Observe that, the leases algorithm does not impose any restriction

[2]The original leases algorithm [8] required that a valid lease be held prior to both read and write requests. For web environments, we assume that clients can only read but not write to an object. Writes to objects are done in a controlled manner directly at the server; so proxies need not implement the part of the leases algorithm that deals with client writes.

on how d is computed. In particular, it does not require the lease duration to be fixed across objects or fixed for a particular object. We exploit this flexibility and propose a number of techniques and policies that balance various tradeoffs. We first present analytical models that use constraints on the state space overhead and control message overhead to compute an appropriate lease duration. A server can employ these models to compute a lease duration based on the observed load; the lease duration can be recomputed if the observed load changes. Since our models are suitable for scenarios where the load does not fluctuate rapidly, we also present policies that *dynamically* determine the lease duration based on the current load. By using the current load conditions to compute the lease duration, these policies can immediately react to load fluctuations. Both techniques enable us to adapt the lease duration to changing conditions, albeit at different time scales. Hence, we collectively refer to our techniques as *adaptive leases*.

III. Analytical Models For Computing the Lease Duration

Consider a web server that stores n data objects and services requests for these objects from proxies as well as end users. Whereas some requests require strong consistency guarantees from the server, weak consistency guarantees suffice for other requests. Let us consider only those requests that require strong consistency guarantees, and let R_i denote the read frequency for object i at a particular proxy p and let W_i denote the its write frequency at the server.[3] Let d_i denote the lease duration, and s_i denote the start time of the lease for object i. Depending on whether the state space at the server or the control messages overhead is the constraining factor, the lease duration can be computed as follows.

A. Lease Duration Based on the State Space Overhead

Let L denote the total state space (in terms of the number of simultaneous leases granted) that the server can maintain. Let λ_i^p denote the frequency of lease grant and renewal messages for object i sent by proxy p. The proxy handles an average of $\lceil d_i \cdot R_i \rceil$ read requests for object i over the duration of its lease. Thus, the cost of a lease grant or renewal request is amortized over $\lceil d_i \cdot R_i \rceil$ read requests at the proxy. Hence, the frequency of lease grant and renewal messages is

$$\lambda_i^p = \frac{R_i}{\lceil d_i \cdot R_i \rceil} \tag{1}$$

The frequency of lease grant/renewal requests for object i received by the server from all proxies is

$$\lambda_i = \sum_p \lambda_i^p \tag{2}$$

[3] A sequence of consecutive writes with no intervening reads is counted as a single write request. This is because, after an invalidating the object from the cache due to the first write, the server need not send additional invalidations for subsequent writes until the proxy fetches the updated object due to a read.

The server grants a lease to each such request and maintains state for the lease over its lifetime $[s_i, s_i + d_i)$. To maintain this state, the server must allocate space by partitioning the total available space L among individual objects. We consider two different policies for doing so.

First, we consider a policy that partitions the state space in proportion to the lease request/renewal frequency (popularity) of an object. Let l_i denote the space (in terms of number of leases) allocated to object i. Then,

$$l_i = \left(\frac{\lambda_i}{\sum_j \lambda_j} \right) \cdot L \tag{3}$$

Since the number of leases granted for object i in the steady state is $d_i \cdot \lambda_i$, we have

$$d_i \cdot \lambda_i \le l_i \tag{4}$$

Substituting l_i from Equation 3 and simplifying, we get,

$$d_i \le \frac{L}{\sum_{j=1}^{n} \lambda_j} \tag{5}$$

Thus, if the available state space is partitioned in proportion to the popularity of an object, then the lease duration for the object is *independent of its lease request/renewal (i.e.,access) frequency* λ_i and depends only on the aggregate request rate at the server. Moreover, the lease duration is *identical for all objects stored at the server.* Observe that such a policy is simple to implement, since the server needs to determine only the aggregate request rate to compute the lease duration and no per-object statistics need to be maintained. The lease duration can be recomputed periodically in case of fluctuations in the aggregate request rate.

Our second policy partitions the available state space equally among all objects. Thus,

$$l_i = \frac{L}{n} \tag{6}$$

Since the number of leases granted to object i in the steady state is $d_i \cdot \lambda_i$, we have

$$d_i \cdot \lambda_i \le l_i = \frac{L}{n} \tag{7}$$

or

$$d_i \le \frac{L}{n \cdot \lambda_i} \tag{8}$$

In this case, the lease duration is inversely proportional to the request/renewal frequency of the object at the server. Thus, *with an equal partitioning of state space across objects, more popular objects are granted shorter leases.*

B. Lease Duration Based on Control Message Overhead

To compute the lease duration based on the control message overhead, we must first quantify the number of messaged exchanged due to read and write requests. Since the proxy handles an average of $\lceil d_i \cdot R_i \rceil$ read requests for object i over the duration of its lease, each lease or renewal request is amortized over $\lceil d_i \cdot R_i \rceil$ reads. Hence the number of control messages per unit time due to read requests is $R_i / \lceil d_i \cdot R_i \rceil$. Each write request results in an invalidation request from the server to the proxy and a subsequent read at the proxy triggers a fetch of the updated object.[4] Thus, each write results in two control messages; the number of control messages per unit time due to write requests is $2W_i$. Observe that, we are only interested in control messages and the overhead of transferring data is not included here. Let C_i denote the bound on the frequency of control messages exchanged for object i between the server and a proxy. Then we have

$$\frac{R_i}{\lceil d_i \cdot R_i \rceil} + 2W_i \leq C_i \tag{9}$$

Substituting $\lceil d_i \cdot R_i \rceil \leq (1 + d_i \cdot R_i)$ and simplifying, we get

$$d_i \geq \frac{1}{C_i - 2W_i} - \frac{1}{R_i} \tag{10}$$

Thus, the lease duration is inversely proportional to the control message overhead and depends on the read frequency R_i at a proxy as well as the write frequency W_i at the server. Furthermore, for a fixed control message overhead, *the more popular an object at a proxy, the longer its lease*.

IV. Adaptive Policies for Computing the Lease Duration

The analytical models presented in Section III enable a server to periodically recompute the lease duration based on the observed load. Such recomputations can be expected to occur over a slow time scale of tens of minutes or hours (since accurate estimates of the load require a large number of samples measured over a long time interval). Consequently, a server employing these models may not be able to react to fast time scale variations in the load. In this section, we present several policies that enable a server to compute the lease duration on-the-fly. Since the lease duration is computed afresh on each lease grant/renewal request, the server can quickly adapt to changing load conditions. Each policy that we present determines the lease duration based on a certain characteristic of the workload and allows a different metric to be optimized.

A. Age-based Leases

This policy is motivated by the bimodal nature of object lifetimes (most objects are long lived, while a majority of the updates go to young objects) [9]. Consequently, a server can reduce the number of invalidate messages it needs to send by granting short leases to frequently modified objects and long leases to long lived objects. Observe that, the policy requires the server to know object lifetimes in order to compute the lease duration. Since lifetimes may not be known a priori, we choose the age of the object to be a reasonable predictor of its lifetime. Hence, the lease duration is computed as

$$d_i = \tau \cdot \mathrm{age}_i \tag{11}$$

where age_i denotes the age of object i and τ is a constant. Our policy assumes that, larger the age of the object, longer is its expected lifetime, and hence, older objects are granted longer leases. However, since the age of an object has no correlation to its popularity [2], old objects that are popular may impose a significant state space overhead on the server. Observe that, this policy is similar to the Alex protocol for computing TTL values [4], [9].

B. Renewal Frequency-based Leases

The policy is motivated by (i) the skewed popularity of objects and (ii) the geographically skewed nature of accesses over the world wide web [9]. A server can exploit these factors to reduce the overhead of lease renewal messages. To do so, the server can grant longer leases to proxies that have a sustained interest in a object. Thus, not only do more popular objects get longer leases, only those proxies at which the object is popular get these long leases. Moreover, granting short leases to proxies that have only a limited interest in the object enables the server to reduce the state space overhead. To achieve these objectives, the lease duration is computed as

$$d_i = \tau \cdot \mathrm{renewal}_i^p \tag{12}$$

where $\mathrm{renewal}_i^p$ denotes the number of renewal messages sent by proxy p for object i, and τ is a constant. A limitation of this policy is that requests for cold objects that were popular in the past continue to be granted long leases. This limitation can be overcome by incorporating another term in Equation 12 that gradually decays the lease duration based on its popularity over a sliding window.

C. State Space Overhead-based Leases

In this policy, the lease duration is set to be inversely proportional to the amount of state maintained at the server. By granting shorter leases during periods of heavy load, the server can adaptively control the amount of state it needs to maintain. The lease duration for an object can be computed either based on the number of valid leases granted for a particular object or the aggregate number of leases granted by the server. That is,

$$d_i = \frac{\tau}{\hat{l}_i} \text{ or } d_i = \frac{\tau}{\hat{L}} \tag{13}$$

where $\hat{l}_i$ and $\hat{L}$ denote the number of leases granted for object i and for all objects at the server, respectively, and τ is a constant.

[4] A sequence of consecutive writes without any intervening reads is counted as a single write, since only a single invalidation message needs to be sent. Also, the derivation assumes $R_i >> W_i$, which implies that a valid lease is always held during writes.

entity-header-extension = lease-control
lease-control = “Lease-Control” “:” lease-directive
lease-directive = lease-request-directive |
lease-response-directive | lease-invalidate-directive |
lease-invalidate-ack-directive
lease-request-directive = “Grant-Lease” | “Renew-Lease”
lease-response-directive = “Lease” “:” lease-period |
“Deny-Lease”
lease-invalidate-directive = “Invalidate-Lease”
lease-invalidate-ack-directive = “Invalidate-Ack” ack
ack = OK | FAILED
lease-period = lease-start “-” lease-expires
lease-start = HTTP-date
lease-expires = HTTP-date

Fig. 2. User-defined extensions to HTTP/1.1 to incorporate leases

V. Prototype Implementation

We have implemented the leases algorithm in the Squid proxy cache and the Apache web server.[5] To do so, we first extended the HTTP/1.1 protocol to enable clients (proxies) to request and renew leases from a server. The HTTP/1.1 protocol allows user defined extensions as part of the request/response header; lease requests and responses use this feature and are piggybacked onto normal http requests and responses. Lease renewals are piggybacked onto if-modified-since http requests. Invalidation requests are also sent as request header extensions. The exact syntax for lease requests, renewals and invalidations is described in Figure 2.

We have incorporated these extensions into the Apache web server (version 1.3.6). Note that, the http protocol (and hence, the Apache web server) is inherently stateless, whereas the leases algorithm requires state to be maintained at the server. Our prototype preserves the stateless nature of the Apache web server by implementing tasks such as granting and renewing leases as well as invalidations in a separate lease server (`leased`). Such an architecture results in a clean separation of functionality between the Apache server, which handles normal http processing, and the lease server which handles lease processing and maintains all the state information (see Figure 3). Whenever the Apache server receives a lease grant/renewal request piggybacked on a http request, it forwards the former to the lease server for further processing. The results of the http request and the lease request are then combined and sent back to the client (proxy). Our lease server implements a number of policies for computing the lease duration; the exact policy to be used can be specified at startup time though a configuration file [7]. Invalidation requests are handled similarly—the web server forwards the request to the lease server, which then sends invalidations to all proxies caching that object. Unreachable proxies are handled by blocking until the lease expires. We have also modified the Squid proxy cache (version 2.2) to support leases. When configured to use leases, our modified Squid proxy sends a lease request with every http request; expired leases are renewed by sending an if-modified-since request to the server (causing the object to be fetched if it has been modified since the lease expiration). Our implementation of leases can coexist with other cache consistency mechanisms such as time-to-live values. The proxy and server can continue to use weak consistency mechanisms such as TTL using normal http request and responses; requests that require strong consistency use http requests/responses enhanced with lease directives.

[5]Source code for our prototype implementation is available from http://www.cs.umass.edu/~lass/software/leases.

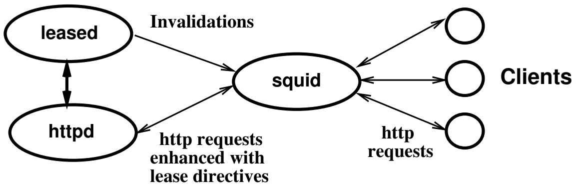


Fig. 3. Interactions between the lease server, the web server and the Squid proxy cache.

VI. Experimental Evaluation

In this section, we demonstrate the efficacy of leases by (i) comparing leases to other cache consistency mechanisms, (ii) evaluating the analytical models presented in Section III, and (iii) evaluating the adaptive lease policies presented in Section IV using trace-driven simulations and the prototype implementation. In what follows, we first present our experimental methodology and then our experimental results.

A. *Experimental Methodology*

A.1 Simulation Environment

We have designed an event based simulator to evaluate the efficacy of various cache consistency mechanisms. The simulator simulates a proxy cache that receives requests from several clients. Cache hits are serviced using locally cached data, whereas a cache miss is simulated by fetching the object from the server. The proxy is assumed to employ a consistency mechanism to ensure the consistency of cached data with that stored on servers. The simulator supports various cache consistency mechanisms such as leases, server-invalidation, client-polling and time-to-live values.

For our experiments, we assume that the proxy employs a disk-based cache to store objects. To determine an appropriate cache size for our experiments, we varied the cache size from 256MB to infinity and found that, for workloads under consideration, the improvements in hit ratios were marginal beyond 1GB. Hence, we choose a disk cache size of 1GB for our experiments so as to factor out the effect of capacity misses. The cache is assumed to be managed using a LRU cache replacement policy. Data retrievals from disk (i.e., cache hits) are modeled using an empirically derived disk model [5] with a fixed OS overhead added to each request. We choose the Seagate Barracuda 4LP disk for parameterizing the disk model [12]. For cache misses, data retrieval time over the network is modeled using the round-trip latency, the network bandwidth, and the object size. Since proxies are deployed close to clients but are distant from most servers, we choose 20 ms and 100KB/s for client-proxy latency and bandwidth, respectively; the proxy-server latency and bandwidth is chosen to be 200ms and 10KB/s (these parameters assume a LAN environment; our results hold for modem environments as well). In reality, network latencies and bandwidths vary depending on network conditions and distance between the source and the destination. Since we are interested in evaluating the efficacy of cache consistency mechanisms, use of a simple network model is adequate for our purpose.

A.2 Workload Characteristics

To generate the workload for our experiments, we use traces from actual proxies, each servicing several thousand clients over a period of several days. We employ three different traces for our experiments; the characteristics of these traces are shown in Table I. To understand the impact of leases on the proxy as well as the server, we determined the most popular server in each trace. We report experimental results for the entire trace (i.e., all servers in the trace) as well as the most popular server.

Each request in the trace provides information such as the time of the request, the requested URL, the size of the object, the client making the request, etc. To determine when objects are modified, we considered using the last modified time values as reported in the trace. However, the BU trace did not include this information, and other traces included these values only when available (e.g., the last modified time time value was unavailable in 37% of the requests contained in the Berkeley trace). Since last modified time values are crucial for evaluating cache consistency mechanisms, we employed an empirically derived model to generate synthetic write requests (and hence, last modified times) for our traces. Based on the observations in [9], we assume that 90% of all web objects change very infrequently (i.e., have an average lifetime of 60 days). We assume that 7% of all objects are mutable (i.e., have an average lifetime of 20 days) and the remaining 3% objects are very mutable (i.e., have an average lifetime of 5 days). We partition all objects in the trace into these three categories[6] and generate write requests and last modified times assuming exponentially distributed lifetimes. The number of synthetic writes generated for each trace is shown in Table I.

B. *Comparison with Other Cache Consistency Schemes*

We first experimentally compare leases with server invalidation and client polling. To do so, we varied the lease duration from 0 to 7 days (the lease duration was kept fixed within each experiment, independent of the workload characteristics) and measured its impact on the server and the proxy. Note that, a lease duration of 0 reduces the scheme to client polling, whereas a lease duration of 7 days (which is larger than the duration of the trace) reduces it to server invalidation. Figure 4 plots the state space overhead, the response time and the control message overhead for all servers in the DEC trace.[7] Figure 5 plots these values for the most popular server in the trace. The figures show that the state space overhead increases with increasing lease duration (since a server must maintain state for each active lease for a longer duration) whereas the control message overhead decreases with increasing lease duration (since the proxy need not poll the server so long as the lease is active). The response time shows a corresponding decrease since cache hits can be serviced without polling the server. For a lease duration of 3600s (1 hour), our technique yields a 425% and 138% improvement in state space and control message overhead, respectively, as compared to server invalidation and client polling (see Figures 4(a) and (c)). Moreover, the degradation in response time as compared to server invalidation is modest at 7.1% (see Figure 4(b)). Figure 4(c) lists various components of the control messages exchanged between the server and the proxy. The figure shows that the number of if-modified-since (IMS) and not-modified messages decreases, whereas the number of invalidation messages increases as we proceed from client polling to server invalidation. The number of GET messages (resulting mostly from compulsory misses) remains relatively unchanged. Since the reduction in IMS and not-modified messages is larger than the increase in invalidation messages, the total control message overhead decreases with increasing lease duration. Together, Figures 4 and 5 demonstrate that by carefully choosing the lease duration a server can tradeoff state space overhead with the number of control messages exchanged, while providing strong consistency guarantees. In the remainder of this section, we study how various policies for computing the lease duration enable a server to make such tradeoffs.

C. *Efficacy of the model*

To evaluate the effectiveness of the model presented in Section III, we first plot the relationship between the state space overhead (i.e., the number of active leases l_i) and the control message overhead C_i. Recall from Eq. 4 and 10, that

$$l_i = \frac{\lambda_i}{C_i - 2W_i} - \frac{\lambda_i}{R_i}$$

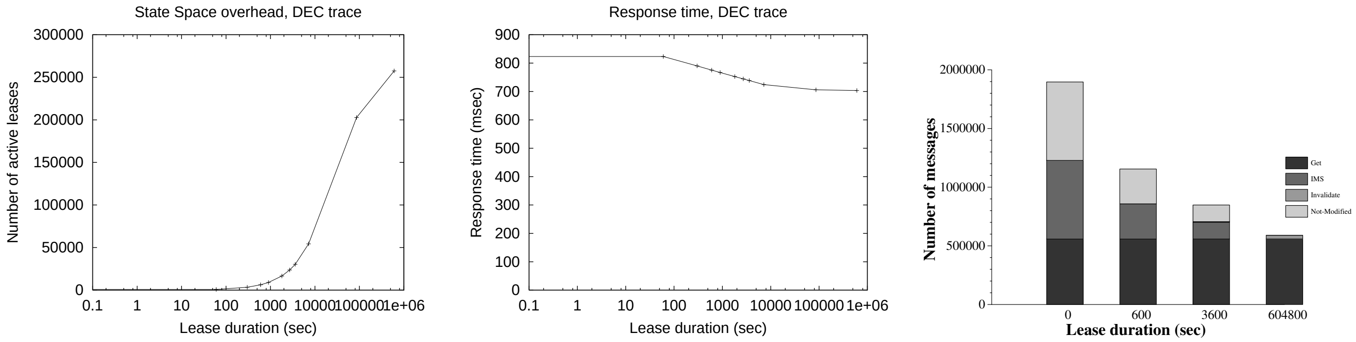
Figure 6 depicts this relationship between l_i and C_i for the most frequently accessed object in the DEC trace. The figure shows that the state space overhead and the control message overhead are inversely proportional to each other. Thus depending on the constraints, a server can trade one

[6] Since a recent study has shown that objects lifetimes and object popularity are uncorrelated [2], we ignored access frequency when partitioning objects into these three categories.

[7] Results from other traces are similar; we omit them due to space constraints.

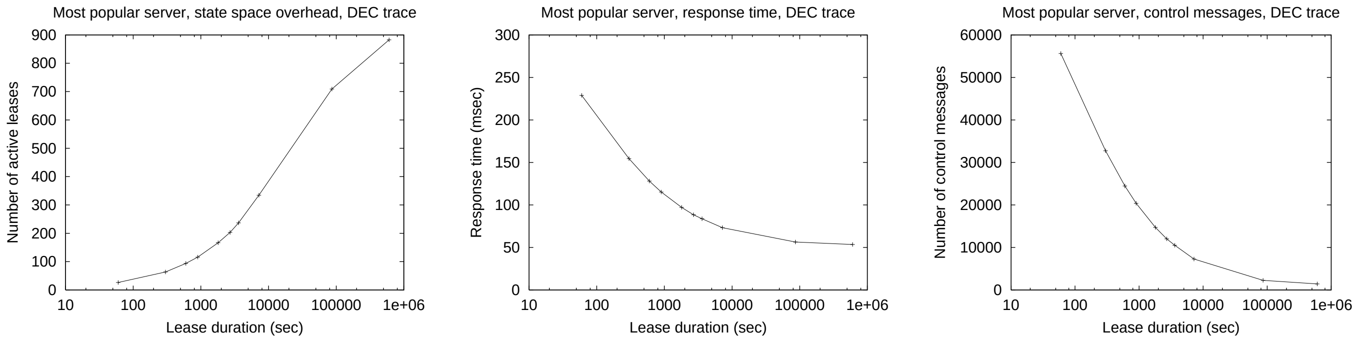
TABLE I
CHARACTERISTICS OF PROXY WORKLOAD TRACES

Trace	Requests	Duration (sec)	Mean request rate (req/s)			Unique URLs	Number of synthetic writes
			Total	Most popular server	Most popular object		
DEC	1228248	146956	8.358	0.2864	0.1565	467593	31267
Berkeley	1000000	154604	6.4	0.0457	0.0258	460292	40274
Boston Univ (BU)	590956	8599470	0.0687	0.0233	0.0116	51136	33515



(a) State Space overhead (b) Response time (c) Control message overhead

Fig. 4. The state space overhead, response times and control message overhead of various cache consistency mechanisms.



(a) State Space overhead (b) Response time (c) Control message overhead

Fig. 5. Performance for the most popular server

for the other by choosing a particular point on this curve, which in turn yields a particular lease duration. Figures 7 and 8 further illustrate this tradeoff. Figure 7(a), obtained from Eq. 4, shows that, for a fixed state space overhead, increasing the aggregate request rate at the server results in a decrease in the lease duration (since the server must grant shorter leases to keep the state space overhead fixed). Figure 7(b) shows that, for a fixed request rate, allocating a larger state space overhead enables a server to proportionately increase the lease duration. The figure also shows that, even for relatively long leases (about 3600s or 1 hour), the state space overhead is modest (1030 leases).

Figure 8(a) and (b), obtained using Eq 10, show the impact of the request rate at a proxy and the control message overhead on the lease duration. Figure 8(a) shows that, for a given C_i, increasing the read request rate R_i causes the lease duration to increase (since the server must grant

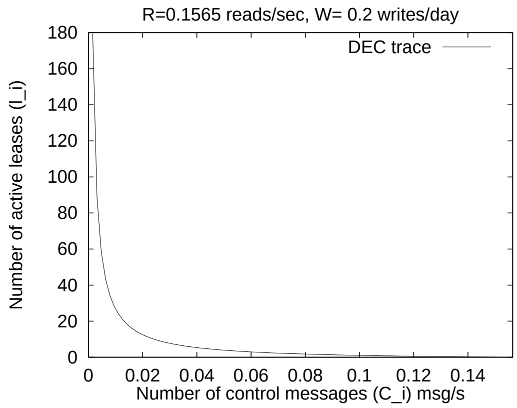
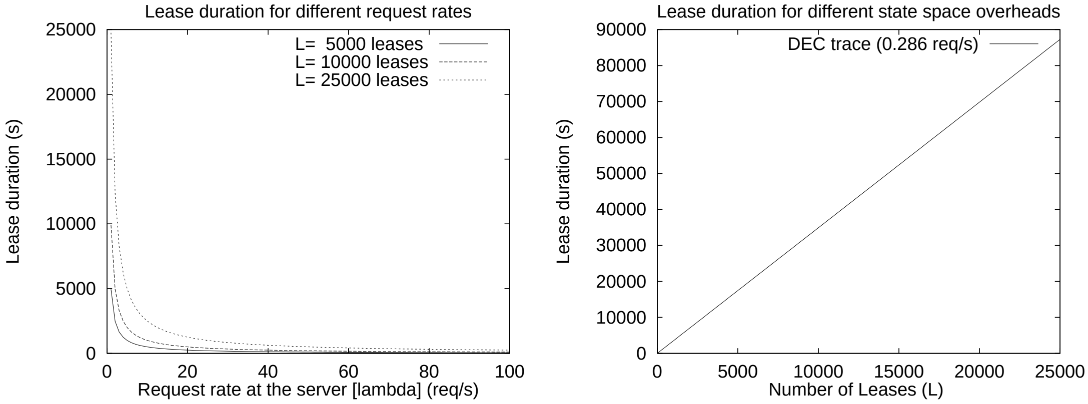


Fig. 6. Relationship between the number of active leases and number of control messages exchanged



(a) (b)

Fig. 7. Effect of request rate and available state space on lease duration.

longer leases to keep the control message overhead fixed). Increasing the control message overhead, on the other hand, enables the server to grant shorter leases (see Figure 8(b)). The figure also shows that for a lease duration of 3600s (1 hour), the control message overhead is 0.0005/s (a message every 33 minutes). Finally, Table II shows the lease durations for various trace workloads obtained using our analytical models. The values shown assume a state space overhead of 1000 leases at the server and a control message overhead of 0.001 msgs/s.

D. Efficacy of Adaptive Leases

To demonstrate the efficacy of the adaptive leases policies presented in Section IV, we varied the average lease duration by varying the parameter τ. Figure 9 plots the state space overhead, control message overhead and response time yielded by each policy for different lease durations. The figure illustrates the following salient features. Renewal-based leases provide longer leases to more popular objects. By doing so, they incur a smaller control message overhead at the expense of a larger state space overhead. For a given τ, the state-space-based lease policy maintains a fixed state space overhead regardless of the load, and hence, the policy incurs the smallest state space overhead (at the expense of a higher control message overhead). Age-based leases neither take the access frequency of an object nor the state space overhead into account. Hence, it incurs the largest control message overhead and response times among the three policies. Together these experiments show that renewal-based leases and state space-based leases are appropriate when the control message overhead and state space overhead, respectively, are constraining factors. As shown in the next section, age-based leases are suitable for optimizing the number of invalidation messages resulting from frequently updated objects.

E. Impact of Object Lifetimes

Figure 10 depicts the impact of object lifetimes on the performance of various adaptive leases policies. Figure 10(a) shows the effect of increasing the object lifetime on the lease duration. Since only the age-based lease policy uses object lifetimes to compute the lease duration, the figure shows that the lease duration increases linearly with increase in object lifetime. The lease duration remains relatively unchanged for the other two policies (since they are independent of object lifetimes). Figure 10(b) shows, for a particular lease duration, that the number of control messages exchanged decreases with increasing object lifetimes. This is because increasing the object lifetime reduces the write frequency for that object and the number of invalidation messages a server must send after each write. Age-based leases exploit this property by granting longer leases to older objects, thereby reducing the number of invalidation messages and state space overhead for frequently updated (young) objects.

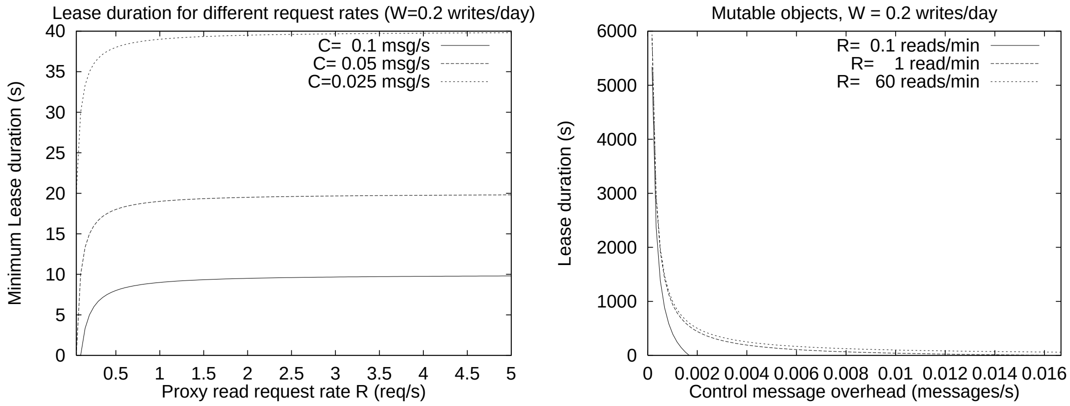
F. Results from the Prototype Implementation

In the preceding sections, we examined the efficacy of various techniques to compute the lease duration. In this section, we study the overhead of granting, renewing and invalidating leases using our prototype implementations. The testbed for our microbencmarks consisted of the augmented web server (httpd and leased), the Squid proxy cache and the client running on a cluster of PC-based workstations. Each PC used in our experiments is a 350MHz Pentium II with 64MB RAM and runs RedHat Linux 5.1; all machines were interconnected by a 10Mb/s ethernet. Our experiment for measuring the overhead of granting a lease consisted of a client that requested a 1KB file first from an unmodified Apache/Squid combination and then from our prototype implementation. The experiment was repeated 2000 times and Table III lists the client response time and the overhead of the leases algorithm at the server. As shown, the overhead of granting a lease is only 3.77ms in our untuned implementation, which is 4.3% of the total response time. Next we compared the overhead of lease renewals to an unmodified Squid that refreshes an expired TTL value using an IMS request. As shown in Table III the overhead of renewing a lease and refreshing an expired TTL

TABLE II

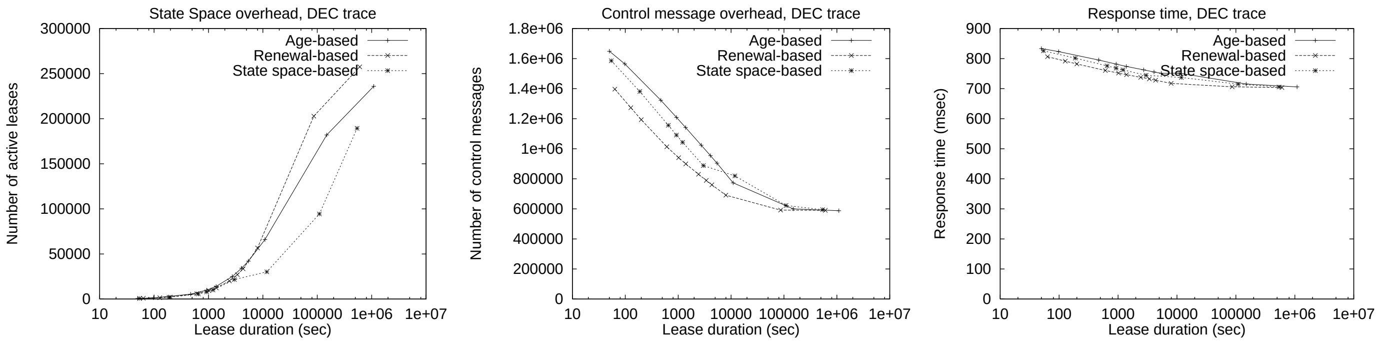
LEASE DURATION COMPUTED USING THE MODEL ($L = 1000, C = 10^{-3}, n = 1000, W_i = 0.2$ WRITES/DAY)

Trace	Request rate at Most Popular Server (λ)	Request rate for Most Popular Object (λ_i)	$d = \frac{L}{\lambda}$	$d = \frac{L}{n\lambda_i}$	$d = \frac{1}{C-2W_i} - \frac{1}{R_i}$
DEC	0.2864	0.01565	58.2 min	53 sec	15.6 min
Berkeley	0.0457	0.02587	360 min	5.47 sec	16.1 min
BU	0.02329	0.01166	715.6 min	85.7 sec	15.31 min



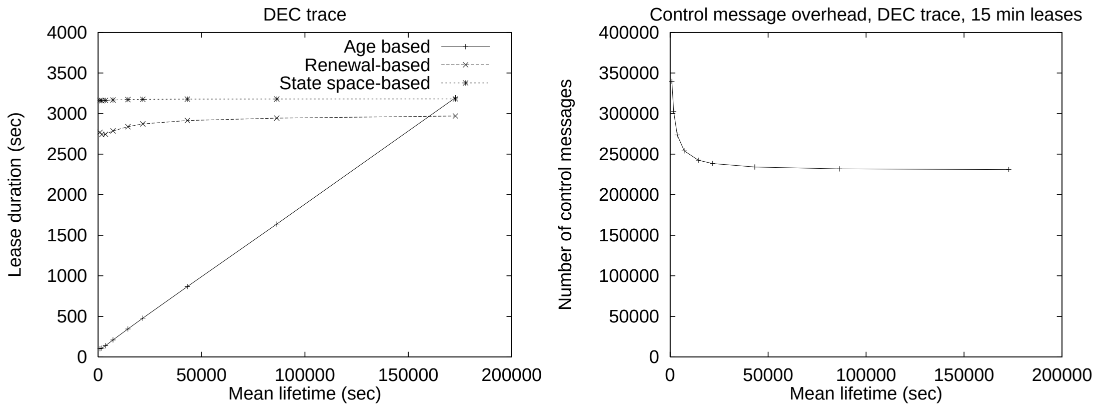
(a) (b)

Fig. 8. Effect of request rate and control message overhead on lease duration.



(a) State Space overhead (b) Control messages (c) Response time

Fig. 9. Performance of adaptive leases algorithm



(a) Duration (b) Control messages

Fig. 10. Impact of write frequency (object lifetimes)

TABLE III
OVERHEAD OF GRANTING AND RENEWING LEASES

	Client response time	Server overhead
Unmodified	76.7ms	-
Grant lease	86.6ms	3.77ms
Renew lease	112.3ms	4.31ms
Refresh TTL	112.0ms	-

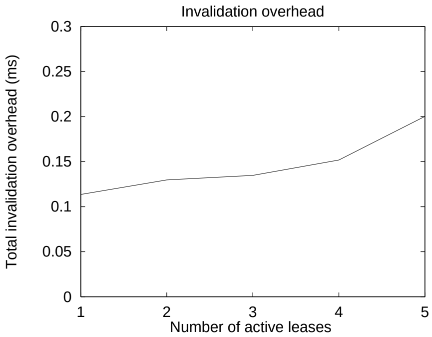


Fig. 11. Overhead of invalidations

value are comparable (the overall response time is larger than the previous case due to the additional computations that Squid performs on a lease/TTL expiration). Figure 11 shows the overhead of sending invalidation messages to proxies. The figure plots the variation in invalidation overhead with increasing number of leases held for an object. Since the lease server must send invalidation messages to each proxy cache holding a valid lease, the invalidation overhead increases slowly with increase in number of active leases (the increase is not linear since the lease server parallelizes this task by forking additional processes). We also measured these overheads by replaying the BU trace and the results were very similar to the microbenchmark results shown in Table III. We omit these results due to space constraints [7]. Thus, the above experiments demonstrate that the leases algorithm can be efficiently implemented with overheads comparable to existing techniques.

VII. CONCLUDING REMARKS

In this paper, we argued that weak cache consistency mechanisms supported by existing web proxy caches must be augmented by strong consistency mechanisms to support the growing diversity in application requirements. Existing strong consistency mechanisms are not appealing for web environments due to their large state space or control message overhead. We focused on the leases approach that balances these tradeoffs and presented analytical models and policies for determining the optimal lease duration. We presented extensions to the http protocol to incorporate leases and then described our prototype implementation using the Squid proxy cache and the Apache web server. Our experimental evaluation of the leases approach showed that: (i) our techniques impose modest overheads even for long leases (a lease duration of 1 hour requires state to be maintained for 1030 leases and imposes an per-object overhead of a control message every 33 minutes); (ii) leases yields a 138-425% improvement over existing strong consistency mechanisms; and (iii) the implementation overhead of leases is comparable to existing weak consistency mechanisms.

REFERENCES

[1] P. Barford, A. Bestavros, A. Bradley, and M. E. Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. *World Wide Web Journal (to appear)*, 1999.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of Infocom'99, New York, NY*, March 1999.

[3] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

[4] V. Cate. Alex: A Global File System. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1–12, May 1992.

[5] P. Chen and D. Patterson. Maximizing Performance in a Striped Disk Array. In *Proceedings of ACM SIGARCH Conference on Computer Architecture, Seattle, WA*, pages 322–331, May 1990.

[6] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. In *Proceedings ACM SIGCOMM'98, Vancouver, BC*, September 1998.

[7] V. Duvvuri. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. Master's thesis, Univ. of Massachusetts, June 1999.

[8] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[9] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[10] B. Krishnamurthy and C. Wills. Proxy Cache Coherency and Replacement—Towards a More Complete Picture. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, June 1999.

[11] J C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of ACM SIGCOMM Conference*, 1997.

[12] Seagate Technology, Inc. *ST-11200N SCSI-2 Fast (Barracuda 4) Specification*, August 1994.

[13] *Squid Internet Object Cache Users Guide*. Available on-line at http://squid.nlanr.net, 1997.

[14] R. Tewari, M. Dahlin, H M. Vin, and J. Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, June 1999.

[15] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of the Usenix Symposium on Internet Technologies (USEITS'99), Boulder, CO (to appear)*, October 1999.

[16] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases for Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, January 1999.

[17] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency Architecture. In *Proceedings of the ACM SIGCOMM'99, Boston, MA (to appear)*, September 1999.